Gulp

中文文档

目錄

gulp 中文文档

> 来源：gulp 中文文档

- 入门指南 - 如何开始使用 gulp
- API 文档 - 学习 gulp 的输入和输出方式
- CLI 文档 - 学习如何执行任务（task）以及如何使用一些编译工具
- 编写插件 - 所以，你已经在写一个 gulp 插件了么？去这儿看一些基本文档，并了解下什么样的事情不应该做

常见问题

常见问题，请查看 FAQ。

秘籍

社区中对于常用的一些 gulp 应用场景已经有了一些现成的秘籍

还是有问题？

到 StackOverflow 发一个带有 #gulp 标签的问题，或者在 Freenode 上的 #gulpjs IRC 频道寻求帮助。

书籍

- Developing a gulp Edge

文章（英文）

- Tagtree intro to gulp video
- Introduction to node.js streams
- Video introduction to node.js streams
- Getting started with gulp (by @markgdyr)
- A cheatsheet for gulp
- Why you shouldn’t create a gulp plugin (or, how to stop worrying and learn to love existing node packages)
- Inspiration (slides) about why gulp was made
- Building With Gulp
- Gulp - The Basics (screencast)
- Get started with gulp (video series)

- Optimize your web code with gulp

例子

- Web Starter Kit gulpfile

License

入门指南

1. 全局安装 gulp：

```
$ npm install --global gulp
```

2. 作为项目的开发依赖（devDependencies）安装：

```
$ npm install --save-dev gulp
```

3. 在项目根目录下创建一个名为 `gulpfile.js` 的文件：

```
var gulp = require('gulp');

gulp.task('default', function() {
  // 将你的默认的任务代码放在这
});
```

4. 运行 gulp：

```
$ gulp
```

默认的名为 default 的任务（task）将会被运行，在这里，这个任务并未做任何事情。

想要单独执行特定的任务（task），请输入 `gulp <task> <othertask>` 。

下一步做什么呢？

你已经安装了所有必要的东西，并且拥有了一个空的 gulpfile。那怎样才算是真的入门了呢？可以查看这些 秘籍 和这个 文章列表 来学习更多的内容。

.src, .watch, .dest, CLI 参数 - 我该怎么去用这些东西呢？

要了解 API 规范文档，请查看 API 文档.

可用的插件

gulp 开发社区正在快速成长，每天都会有新的插件诞生。在 主站 上可以查看完整的列表。

gulp API 文档

gulp.src(globs[, options])

输出（Emits）符合所提供的匹配模式（glob）或者匹配模式的数组（array of globs）的文件。将返回一个 Vinyl files 的 stream 它可以被 piped 到别的插件中。

```
gulp.src('client/templates/*.jade')
  .pipe(jade())
  .pipe(minify())
  .pipe(gulp.dest('build/minified_templates'));
```

`glob` 请参考 node-glob 语法 或者，你也可以直接写文件的路径。

globs

类型： `String` 或 `Array`

所要读取的 glob 或者包含 globs 的数组。

options

类型： `Object`

通过 glob-stream 所传递给 node-glob 的参数。

除了 node-glob 和 glob-stream 所支持的参数外，gulp 增加了一些额外的选项参数：

options.buffer

类型： `Boolean` 默认值： `true`

如果该项被设置为 `false` ，那么将会以 stream 方式返回 `file.contents` 而不是文件 buffer 的形式。这在处理一些大文件的时候将会很有用。注意：插件可能并不会实现对 stream 的支持。

options.read

类型： `Boolean` 默认值： `true`

如果该项被设置为 `false` ， 那么 `file.contents` 会返回空值（null），也就是并不会去读取文件。

options.base

类型： `String` 默认值： 将会加在 glob 之前 (请看 glob2base)

如, 请想像一下在一个路径为 `client/js/somedir` 的目录中，有一个文件叫 `somefile.js` ：

```
gulp.src('client/js/**/*.js') // 匹配 'client/js/somedir/somefile.j
  .pipe(minify())
  .pipe(gulp.dest('build'));  // 写入 'build/somedir/somefile.js'

gulp.src('client/js/**/*.js', { base: 'client' })
  .pipe(minify())
  .pipe(gulp.dest('build'));  // 写入 'build/js/somedir/somefile.js
```

gulp.dest(path[, options])

能被 pipe 进来，并且将会写文件。并且重新输出（emits）所有数据，因此你可以将它 pipe 到多个文件夹。如果某文件夹不存在，将会自动创建它。

```
gulp.src('./client/templates/*.jade')
  .pipe(jade())
  .pipe(gulp.dest('./build/templates'))
  .pipe(minify())
  .pipe(gulp.dest('./build/minified_templates'));
```

文件被写入的路径是以所给的相对路径根据所给的目标目录计算而来。类似的，相对路径也可以根据所给的 base 来计算。 请查看上述的 `gulp.src` 来了解更多信息。

path

类型： `String` or `Function`

文件将被写入的路径（输出目录）。也可以传入一个函数，在函数中返回相应路径，这个函数也可以由 vinyl 文件实例 来提供。

options

类型： `Object`

options.cwd

类型： `String` 默认值： `process.cwd()`

输出目录的 `cwd` 参数，只在所给的输出目录是相对路径时候有效。

options.mode

类型： `String` 默认值： `0777`

八进制权限字符，用以定义所有在输出目录中所创建的目录的权限。

gulp.task(name[, deps], fn)

定义一个使用 Orchestrator 实现的任务（task）。

```
gulp.task('somename', function() {
  // 做一些事
});
```

name

任务的名字，如果你需要在命令行中运行你的某些任务，那么，请不要在名字中使用空格。

deps

类型： `Array`

一个包含任务列表的数组，这些任务会在你当前任务运行之前完成。

```
gulp.task('mytask', ['array', 'of', 'task', 'names'], function() {
  // 做一些事
});
```

注意： 你的任务是否在这些前置依赖的任务完成之前运行了？请一定要确保你所依赖的任务列表中的任务都使用了正确的异步执行方式：使用一个 callback，或者返回一个 promise 或 stream。

fn

该函数定义任务所要执行的一些操作。通常来说，它会是这种形式： `gulp.src().pipe(someplugin())` 。

异步任务支持

任务可以异步执行，如果 `fn` 能做到以下其中一点：

接受一个 callback

```
// 在 shell 中执行一个命令
var exec = require('child_process').exec;
gulp.task('jekyll', function(cb) {
  // 编译 Jekyll
  exec('jekyll build', function(err) {
    if (err) return cb(err); // 返回 error
    cb(); // 完成 task
  });
});
```

返回一个 stream

```
gulp.task('somename', function() {
  var stream = gulp.src('client/**/*.js')
    .pipe(minify())
    .pipe(gulp.dest('build'));
  return stream;
});
```

返回一个 promise

```
var Q = require('q');

gulp.task('somename', function() {
  var deferred = Q.defer();

  // 执行异步的操作
  setTimeout(function() {
    deferred.resolve();
  }, 1);

  return deferred.promise;
});
```

注意： 默认的，task 将以最大的并发数执行，也就是说，gulp 会一次性运行所有的 task 并且不做任何等待。如果你想要创建一个序列化的 task 队列，并以特定的顺序执行，你需要做两件事：

- 给出一个提示，来告知 task 什么时候执行完毕，
- 并且再给出一个提示，来告知一个 task 依赖另一个 task 的完成。

对于这个例子，让我们先假定你有两个 task，"one" 和 "two"，并且你希望它们按照这个顺序执行：

1. 在 "one" 中，你加入一个提示，来告知什么时候它会完成：可以再完成时候返回一个 callback，或者返回一个 promise 或 stream，这样系统会去等待它完成。
2. 在 "two" 中，你需要添加一个提示来告诉系统它需要依赖第一个 task 完成。

因此，这个例子的实际代码将会是这样：

```
var gulp = require('gulp');

// 返回一个 callback，因此系统可以知道它什么时候完成
gulp.task('one', function(cb) {
    // 做一些事 -- 异步的或者其他的
    cb(err); // 如果 err 不是 null 或 undefined，则会停止执行，且注意，这
});

// 定义一个所依赖的 task 必须在这个 task 执行之前完成
gulp.task('two', ['one'], function() {
    // 'one' 完成后
});

gulp.task('default', ['one', 'two']);
```

gulp.watch(glob [, opts], tasks) 或 gulp.watch(glob [, opts, cb])

监视文件，并且可以在文件发生改动时候做一些事情。它总会返回一个 EventEmitter 来发射（emit） `change` 事件。

gulp.watch(glob[, opts], tasks)

glob

类型： `String` or `Array`

一个 glob 字符串，或者一个包含多个 glob 字符串的数组，用来指定具体监控哪些文件的变动。

opts

类型： `Object`

传给 `gaze` 的参数。

tasks

类型： `Array`

需要在文件变动后执行的一个或者多个通过 `gulp.task()` 创建的 task 的名字，

```
var watcher = gulp.watch('js/**/*.js', ['uglify','reload']);
watcher.on('change', function(event) {
  console.log('File ' + event.path + ' was ' + event.type + ', runn
});
```

gulp.watch(glob[, opts, cb])

glob

类型： `String` or `Array`

一个 glob 字符串，或者一个包含多个 glob 字符串的数组，用来指定具体监控哪些文件的变动。

opts

类型： `Object`

传给 `gaze` 的参数。

cb(event)

类型： `Function`

每次变动需要执行的 callback。

```
gulp.watch('js/**/*.js', function(event) {
  console.log('File ' + event.path + ' was ' + event.type + ', runn
});
```

callback 会被传入一个名为 `event` 的对象。这个对象描述了所监控到的变动：

event.type

类型： `String`

发生的变动的类型： `added` , `changed` 或者 `deleted` 。

event.path

类型： `String`

触发了该事件的文件的路径。

gulp 命令行（CLI）文档

参数标记

gulp 只有你需要熟知的参数标记，其他所有的参数标记只在一些任务需要的时候使用。

- `-v` 或 `--version` 会显示全局和项目本地所安装的 gulp 版本号
- `--require <module path>` 将会在执行之前 reqiure 一个模块。这对于一些语言编译器或者需要其他应用的情况来说来说很有用。你可以使用多个 `--require`
- `--gulpfile <gulpfile path>` 手动指定一个 gulpfile 的路径，这在你有很多个 gulpfile 的时候很有用。这也会将 CWD 设置到该 gulpfile 所在目录
- `--cwd <dir path>` 手动指定 CWD。定义 gulpfile 查找的位置，此外，所有的相应的依赖（require）会从这里开始计算相对路径
- `-T` 或 `--tasks` 会显示所指定 gulpfile 的 task 依赖树
- `--tasks-simple` 会以纯文本的方式显示所载入的 gulpfile 中的 task 列表
- `--color` 强制 gulp 和 gulp 插件显示颜色，即便没有颜色支持
- `--no-color` 强制不显示颜色，即便检测到有颜色支持
- `--silent` 禁止所有的 gulp 日志

命令行会在 process.env.INIT_CW 中记录它是从哪里被运行的。

Task 特定的参数标记

请参考 StackOverflow 了解如何增加任务特定的参数标记。

Tasks

Task 可以通过 `gulp <task> <othertask>` 方式来执行。如果只运行 `gulp` 命令，则会执行所注册的名为 `default` 的 task，如果没有这个 task，那么 gulp 会报错。

编译器

你可以在 interpret 找到所支持的语言列表。如果你想要增加一个语言的支持，请在这里提交一个 pull request 或者 issue。

编写插件

如果你打算自己写一个 Gulp 插件，为了节约你的时间，你可以先完整地阅读下这个文档。

- 导览 (必读)
- 使用 buffer
- 使用 stream 来处理
- 测试

它要做什么？

流式处理文件对象（Streaming file objects）

gulp 插件总是返回一个 object mode 形式的 stream 来做这些事情：

1. 接收 vinyl File 对象
2. 输出 vinyl File 对象

这通常被叫做 transform streams (有时候也叫做 through streams)。transform streams 是可读又可写的，它会对传给它的对象做一些转换的操作。

修改文内容

Vinyl 文件可以通过三种不同形式来访问文件内容：

- Streams
- Buffers
- 空 (null) - 对于删除, 清理, 等操作来说，会很有用，因为这时候内容是不需要处理的。

有用的资源

- File object
- PluginError
- event-stream
- BufferStream
- gulp-util

插件范例

- sindresorhus' gulp plugins
- Fractal's gulp plugins

- gulp-replace

关于 stream

如果你不熟悉 stream，你可以阅读这些来

- https://github.com/substack/stream-handbook (必读)
- http://nodejs.org/api/stream.html

其他的一些为 gulp 创建的和使用的，但又并非通过 stream 去处理的库，在 npm 上都会被打上 gulpfriendly 标签。

指导

> 这个指导实际上不是必须的，但是我们强烈建议每一个人来遵守。因为没有人会喜欢用一个不好的插件。这个指导能在某种意义上确保你的插件能很好的适应 gulp，以此来让你的生活变得更将轻松。

编写插件 > 指导

1. 你的插件不应该去做一些现有 node 模块已经能很容易做到的事情
2. 比如：删除一个文件夹并不需要做成一个 gulp 插件，在 task 里使用一个类似 del 这样的插件即可。
3. 只是为了封装而封装一些的东西进去，这只会增加很多低质量的插件到生态中，这不符合 gulp 的期望。
4. gulp 插件都是以文件为基础操作的，如果你发现你正在把一些很复杂的操作塞进 stream 中去，那么，请直接写一个 node 模块就好。
5. 一个好的 gulp 插件例子像是 gulp-coffee，coffee-script 模块并不能直接和 vinyl 做很好的适配，因此，才去封装它来使用相应的功能，并且将一些比较痛苦的操作抽象出来，做成更简单的 gulp 插件来使用。
6. 你的插件应该只做一件事，并且做好。
7. 避免使用配置选项，使得你的插件能胜任不同场合的任务。
8. 比如：一个 JS 压缩插件不应该有一个加头部的选项
9. 你的插件不能去做一些其他插件做的事：
10. 不应该去拼接，用 gulp-concat 去做
11. 不应该去增加头部，用 gulp-header 去做
12. 不应该去增加尾部，用 gulp-footer 去做
13. 如果是一个常用的可选的操作，那么，请在文档中注明你的插件通常和其他某个插件一起使用
14. 在你的插件中使用其他的插件，这能大大减少你的代码量，并保证生态系统的稳定。
15. 你的插件必须被测试过
16. 测试一个插件很简单，你甚至不需要 gulp 就能测试
17. 参考其他的插件是怎么做的
18. 在 `package.json` 中增加一个名为 `gulpplugin` 的关键字，这可以让它能在我们的搜索中出现
19. 不要再 stream 里面抛出错误
20. 你应该以触发 **error** 事件来代替

21. 如果你在 stream 外面遇到错误，比如在创建 stream 时候发现错误的配置选项等，那么你应该抛出它。
22. 错误需要加上以你插件名字作为前缀
23. 比如： `gulp-replace: Cannot do regexp replace on a stream`
24. 使用 gulp-util 的 PluginError 类来完成它
25. `file.contents` 的类型需要总是在输入输出中保持一致
26. 如果 file.contents 为空 (不可读) 请将他忽略，并传过去
27. 如果 file.contents 是一个 stream，但是你不支持，那么请触发一个错误
    - 不要把 stream 硬转成 buffer 来使你的插件支持 stream，这会引发很严重的问题。
28. 在你处理完成之前，不要将 `file` 传到下游去
29. 使用 `file.clone()` 来复制一个文件或者创建另一个以此为基础的文件
30. 使用我们 模块推荐页 上列举的模块来让你的开发更加轻松
31. 不要把 `gulp` 作为一个依赖
32. 使用 gulp 来测试你的插件的工作流这的确很酷，但请务必确保你将它放到 devDependency 中
33. 在你的插件中依赖 gulp，这意味着安装你的插件的用户将会重新安装一遍 gulp 以及所有它所依赖的东西。
34. 没有任何理由说明你需要将 gulp 写到你的插件代码中去，如果你发现你必须这么做，那么请开一个 issue，我们会帮你解决。

为什么这些指导这么严格？

gulp 的目标是为了让用户觉得简单，通过提供一些严格的指导，我们就能提供一致并且高质量的生态系统给大家。不过，这确实给插件作者增加了一些需要考虑的东西，但是也确保了后面的问题会更少。

如果我不遵守这些，会发生什么？

npm 对每个人来说是免费的，你可以开发任何你想要开发的东西出来，并且不需要遵守这个规定。我们承诺测试机制将会很快建立起来，并且加入我们的插件搜索中。如果你坚持不遵守插件导览，那么这会反应在我们的打分/排名系统上，人们都会更加喜欢去使用一个 "更加 gulp" 的插件。

一个插件大概会是怎么样的？

```
// through2 是一个对 node 的 transform streams 简单封装
var through = require('through2');
var gutil = require('gulp-util');
var PluginError = gutil.PluginError;

// 常量
const PLUGIN_NAME = 'gulp-prefixer';

function prefixStream(prefixText) {
  var stream = through();
  stream.write(prefixText);
  return stream;
}

// 插件级别函数 (处理文件)
function gulpPrefixer(prefixText) {

  if (!prefixText) {
    throw new PluginError(PLUGIN_NAME, 'Missing prefix text!');
  }
  prefixText = new Buffer(prefixText); // 预先分配

  // 创建一个让每个文件通过的 stream 通道
  return through.obj(function(file, enc, cb) {
    if (file.isNull()) {
      // 返回空文件
      cb(null, file);
    }
    if (file.isBuffer()) {
      file.contents = Buffer.concat([prefixText, file.contents]);
    }
    if (file.isStream()) {
      file.contents = file.contents.pipe(prefixStream(prefixText))
    }

    cb(null, file);

  });

};

// 暴露（export）插件主函数
module.exports = gulpPrefixer;
```

使用 buffer

> 这里有些关于如何创建一个使用 buffer 来处理的插件的有用信息。

编写插件 > 使用 buffer

使用 buffer

如果你的插件依赖着一个基于 buffer 处理的库，你可能会选择让你的插件以 buffer 的形式来处理 file.contents。让我们来实现一个在文件头部插入额外文本的插件：

```
var through = require('through2');
var gutil = require('gulp-util');
var PluginError = gutil.PluginError;

// 常量
const PLUGIN_NAME = 'gulp-prefixer';

// 插件级别的函数（处理文件）
function gulpPrefixer(prefixText) {
  if (!prefixText) {
    throw new PluginError(PLUGIN_NAME, 'Missing prefix text!');
  }

  prefixText = new Buffer(prefixText); // 提前分配

  // 创建一个 stream 通道，以让每个文件通过
  var stream = through.obj(function(file, enc, cb) {
    if (file.isStream()) {
      this.emit('error', new PluginError(PLUGIN_NAME, 'Streams are
      return cb();
    }

    if (file.isBuffer()) {
      file.contents = Buffer.concat([prefixText, file.contents]);
    }

    // 确保文件进入下一个 gulp 插件
    this.push(file);

    // 告诉 stream 引擎，我们已经处理完了这个文件
    cb();
  });

  // 返回文件 stream
  return stream;
};

// 导出插件主函数
module.exports = gulpPrefixer;
```

上述的插件可以这样使用：

```
var gulp = require('gulp');
var gulpPrefixer = require('gulp-prefixer');

gulp.src('files/**/*.js')
  .pipe(gulpPrefixer('prepended string'))
  .pipe(gulp.dest('modified-files'));
```

处理 stream

不幸的是，当 gulp.src 如果是以 stream 的形式，而不是 buffer，那么，上面的插件就会报错。如果可以，你也应该让他支持 stream 形式。请查看使用 Stream 处理获取更多信息。

一些基于 buffer 的插件

- gulp-coffee
- gulp-svgmin
- gulp-marked
- gulp-svg2ttf

使用 Stream 处理

> 极力推荐让你所写的插件支持 stream。这里有一些关于让插件支持 stream 的一些有用信息。
>
> 请确保使用处理错误的最佳实践，并且加入一行代码，使得 gulp 能在转换内容的期间在捕获到第一个错误时候正确报出错误。

编写插件 > 编写以 stream 为基础的插件

使用 stream 处理

让我们来实现一个用于在文件头部插入一些文本的插件，这个插件支持 file.contents 所有可能的形式。

```
var through = require('through2');
var gutil = require('gulp-util');
var PluginError = gutil.PluginError;

// 常量
const PLUGIN_NAME = 'gulp-prefixer';

function prefixStream(prefixText) {
  var stream = through();
  stream.write(prefixText);
  return stream;
}

// 插件级别函数 (处理文件)
function gulpPrefixer(prefixText) {
  if (!prefixText) {
    throw new PluginError(PLUGIN_NAME, 'Missing prefix text!');
  }

  prefixText = new Buffer(prefixText); // 预先分配

  // 创建一个让每个文件通过的 stream 通道
  var stream = through.obj(function(file, enc, cb) {
    if (file.isBuffer()) {
      this.emit('error', new PluginError(PLUGIN_NAME, 'Buffers not
      return cb();
    }

    if (file.isStream()) {
      // 定义转换内容的 streamer
      var streamer = prefixStream(prefixText);
      // 从 streamer 中捕获错误，并发出一个 gulp的错误
      streamer.on('error', this.emit.bind(this, 'error'));
      // 开始转换
      file.contents = file.contents.pipe(streamer);
    }

    // 确保文件进去下一个插件
    this.push(file);
    // 告诉 stream 转换工作完成
    cb();
  });

  // 返回文件 stream
  return stream;
}

// 暴露（export）插件的主函数
module.exports = gulpPrefixer;
```

上面的插件可以像这样使用：

```
var gulp = require('gulp');
var gulpPrefixer = require('gulp-prefixer');

gulp.src('files/**/*.js', { buffer: false })
  .pipe(gulpPrefixer('prepended string'))
  .pipe(gulp.dest('modified-files'));
```

一些使用 stream 的插件

- gulp-svgicons2svgfont

测试

> 测试是保证你的插件质量的唯一途径。这能使你的用户有信心去使用，且能让你更加轻松。

编写插件 > 测试

工具

大多数的插件使用 mocha，should 以及 event-stream 来做测试。下面的例子也将会使用这些工具。

测试插件的流处理（streaming）模式

```
var assert = require('assert');
var es = require('event-stream');
var File = require('vinyl');
var prefixer = require('../');

describe('gulp-prefixer', function() {
  describe('in streaming mode', function() {

    it('should prepend text', function(done) {

      // 创建伪文件
      var fakeFile = new File({
        contents: es.readArray(['stream', 'with', 'those', 'content
      });

      // 创建一个 prefixer 流（stream）
      var myPrefixer = prefixer('prependthis');

      // 将伪文件写入
      myPrefixer.write(fakeFile);

      // 等文件重新出来
      myPrefixer.once('data', function(file) {
        // 确保它以相同的方式出来
        assert(file.isStream());

        // 缓存内容来确保它已经被处理过（加前缀内容）
        file.contents.pipe(es.wait(function(err, data) {
          // 检查内容
          assert.equal(data, 'prependthisstreamwiththosecontents')
          done();
        }));
      });

    });

  });
});
```

测试插件的 buffer 模式

```
var assert = require('assert');
var es = require('event-stream');
var File = require('vinyl');
var prefixer = require('../');

describe('gulp-prefixer', function() {
  describe('in buffer mode', function() {

    it('should prepend text', function(done) {

      // 创建伪文件
      var fakeFile = new File({
        contents: new Buffer('abufferwiththiscontent')
      });

      // 创建一个 prefixer 流（stream）
      var myPrefixer = prefixer('prependthis');

      // 将伪文件写入
      myPrefixer.write(fakeFile);

      // 等文件重新出来
      myPrefixer.once('data', function(file) {
        // 确保它以相同的方式出来
        assert(file.isBuffer());

        // 检查内容
        assert.equal(file.contents.toString('utf8'), 'prependthisal
        done();
      });

    });

  });
});
```

一些拥有高质量的测试用例的插件

- gulp-cat
- gulp-concat

FAQ

为什用 gulp 而不是 ____?

请先看 gulp 介绍幻灯片 来大致了解下 gulp 是怎么来的。

是 "gulp" 还是 "Gulp"？

gulp 一直都是小写的。除了在 gulp 的 logo 中是用大写的。

去哪里可以找到 gulp 插件的列表？

gulp 插件总是会包含 `gulpplugin` 关键字。在这搜索 gulp 插件 或者 在 npm 查看所有插件。

我想写一个 gulp 插件，我应该从哪里开始呢？

请查看 编写插件 wiki 页面来阅读一些指导以及一些例子。

我的插件将做 ____, 它是不是做的太多了？

有可能。可以先自问下：

1. 我的插件是否做了一些其他插件可能需要做的事情？
2. 如果是，那么那一段功能应该作为一个独立的插件。查看是否已经有相应的插件存在了.
3. 我的插件是否做了两件事，两件根据配置的不同而截然不同的事情？
4. 如果是，那么为了社区的良好发展，最好是分开为两个插件发布
5. 如果两个任务是不同的，但是差别非常细微，那实际上是允许的

换行符在插件输出中应该如何表示？

请总是使用 `\n` 以避免不同的操作系统带来的兼容性问题。

我可以从哪里获取 gulp 的最新信息？

gulp 的更新信息可以通过关注以下的 twitter 来获取：

- @wearefractal
- @eschoff
- @gulpjs

gulp 是否有 IRC 频道？

有的，欢迎来 Freenode 上的 #gulpjs 来交流。

gulp 技巧集

- 整合 streams 来处理错误
- 删除文件和文件夹
- 使用 watchify 加速 browserify 编译
- 增量编译打包，包括处理整所涉及的所有文件
- 将 buffer 变为 stream (内存中的内容)
- 在 gulp 中运行 Mocha 测试
- 仅仅传递更改过的文件
- 从命令行传递参数
- 只重新编译被更改过的文件
- 每个文件夹生成单独一个文件
- 串行方式运行任务
- 拥有实时重载（live-reloading）和 CSS 注入的服务器
- 通过 stream 工厂来共享 stream
- 指定一个新的 cwd (当前工作目录)
- 分离任务到多个文件中
- 使用外部配置文件
- 在一个任务中使用多个文件来源
- Browserify + Uglify2 和 sourcemaps
- Browserify + Globs
- 同时输出一个压缩过和一个未压缩版本的文件
- 改变版本号以及创建一个 git tag
- Swig 以及 YAML front-matter 模板

整合 streams 来处理错误

默认情况下，在 stream 中发生一个错误的话，它会被直接抛出，除非已经有一个时间监听器监听着 `error` 时间。 这在处理一个比较长的管道操作的时候会显得比较棘手。

通过使用 stream-combiner2，你可以将一系列的 stream 合并成一个，这意味着，你只需要在你的代码中一个地方添加监听器监听 `error` 时间就可以了。

这里是一个在 gulpfile 中使用它的例子：

```
var combiner = require('stream-combiner2');
var uglify = require('gulp-uglify');
var gulp = require('gulp');

gulp.task('test', function() {
  var combined = combiner.obj([
    gulp.src('bootstrap/js/*.js'),
    uglify(),
    gulp.dest('public/bootstrap')
  ]);

  // 任何在上面的 stream 中发生的错误，都不会抛出，
  // 而是会被监听器捕获
  combined.on('error', console.error.bind(console));

  return combined;
});
```

删除文件和文件夹

你也许会想要在编译文件之前删除一些文件。由于删除文件和文件内容并没有太大关系，所以，我们没必要去用一个 gulp 插件。最好的一个选择就是使用一个原生的 node 模块。

因为 `del` 模块支持多个文件以及 globbing，因此，在这个例子中，我们将使用它来删除文件：

```
$ npm install --save-dev gulp del
```

假想有如下的文件结构：

```
.
├── dist
│   ├── report.csv
│   ├── desktop
│   └── mobile
│       ├── app.js
│       ├── deploy.json
│       └── index.html
└── src
```

在 gulpfile 中，我们希望在运行我们的编译任务之前，将 `mobile` 文件的内容先清理掉：

```
var gulp = require('gulp');
var del = require('del');

gulp.task('clean:mobile', function (cb) {
  del([
    'dist/report.csv',
    // 这里我们使用一个通配模式来匹配 `mobile` 文件夹中的所有东西
    'dist/mobile/**/*',
    // 我们不希望删掉这个文件，所以我们取反这个匹配模式
    '!dist/mobile/deploy.json'
  ], cb);
});

gulp.task('default', ['clean:mobile']);
```

在管道中删除文件

你可能需要在管道中将一些处理过的文件删除掉。

我们使用 vinyl-paths 模块来简单地获取 stream 中每个文件的路径，然后传给 `del` 方法。

```
$ npm install --save-dev gulp del vinyl-paths
```

假想有如下的文件结构：

```
.
├── tmp
│   ├── rainbow.js
│   └── unicorn.js
└── dist
```

```
var gulp = require('gulp');
var stripDebug = require('gulp-strip-debug'); // 仅用于本例做演示
var del = require('del');
var vinylPaths = require('vinyl-paths');

gulp.task('clean:tmp', function () {
  return gulp.src('tmp/*')
    .pipe(stripDebug())
    .pipe(gulp.dest('dist'))
    .pipe(vinylPaths(del));
});

gulp.task('default', ['clean:tmp']);
```

只有在已经使用了其他的插件之后才需要这样做，否则，请直接使用 `gulp.src` 来代替。

使用 watchify 加速 browserify 编译

当一个 browserify 项目开始变大的时候，编译打包的时间也会慢慢变得长起来。虽然开始的时候可能只需花 1 秒，然后当你的项目需要建立在一些流行的大型项目的基础上时，它很有可能就变成 30 秒了。

这就是为什么 substack 写了 watchify 的原因，一个持续监视文件的改动，并且 只重新打包必要的文件 的 browserify 打包工具。用这种方法，第一次打包的时候可能会还是会花 30 秒，但是后续的编译打包工作将一直保持在 100 毫秒以下 —— 这是一个极大的提升。

watchify 并没有一个相应的 gulp 插件，并且也不需要有：你可以使用 vinyl-source-stream 来把你的用于打包的 stream 连接到 gulp 管道中。

```
'use strict';

var watchify = require('watchify');
var browserify = require('browserify');
var gulp = require('gulp');
var source = require('vinyl-source-stream');
var buffer = require('vinyl-buffer');
var gutil = require('gulp-util');
var sourcemaps = require('gulp-sourcemaps');
var assign = require('lodash.assign');

// 在这里添加自定义 browserify 选项
var customOpts = {
  entries: ['./src/index.js'],
  debug: true
};
var opts = assign({}, watchify.args, customOpts);
var b = watchify(browserify(opts));

// 在这里加入变换操作
// 比如： b.transform(coffeeify);

gulp.task('js', bundle); // 这样你就可以运行 `gulp js` 来编译文件了
b.on('update', bundle); // 当任何依赖发生改变的时候，运行打包工具
b.on('log', gutil.log); // 输出编译日志到终端

function bundle() {
  return b.bundle()
    // 如果有错误发生，记录这些错误
    .on('error', gutil.log.bind(gutil, 'Browserify Error'))
    .pipe(source('bundle.js'))
    // 可选项，如果你不需要缓存文件内容，就删除
    .pipe(buffer())
    // 可选项，如果你不需要 sourcemaps，就删除
    .pipe(sourcemaps.init({loadMaps: true})) // 从 browserify 文件载
       // 在这里将变换操作加入管道
    .pipe(sourcemaps.write('./')) // 写入 .map 文件
    .pipe(gulp.dest('./dist'));
}
```

增量编译打包，包括处理整所涉及的所有文件

在做增量编译打包的时候，有一个比较麻烦的事情，那就是你常常希望操作的是 所有 处理过的文件，而不仅仅是单个的文件。举个例子，你想要只对更改的文件做代码 lint 操作，以及一些模块封装的操作，然后将他们与其他已经 lint 过的，以及已经进行过模块封装的文件合并到一起。如果不用到临时文件的话，这将会非常困难。

使用 gulp-cached 以及 gulp-remember 来解决这个问题。

```
var gulp = require('gulp');
var header = require('gulp-header');
var footer = require('gulp-footer');
var concat = require('gulp-concat');
var jshint = require('gulp-jshint');
var cached = require('gulp-cached');
var remember = require('gulp-remember');

var scriptsGlob = 'src/**/*.js';

gulp.task('scripts', function() {
  return gulp.src(scriptsGlob)
      .pipe(cached('scripts'))        // 只传递更改过的文件
      .pipe(jshint())                 // 对这些更改过的文件做一些特殊的
      .pipe(header('(function () {')) // 比如 jshinting ^^^
      .pipe(footer('})();'))          // 增加一些类似模块封装的东西
      .pipe(remember('scripts'))      // 把所有的文件放回 stream
      .pipe(concat('app.js'))         // 做一些需要所有文件的操作
      .pipe(gulp.dest('public/'));
});

gulp.task('watch', function () {
  var watcher = gulp.watch(scriptsGlob, ['scripts']); // 监视与 scri
  watcher.on('change', function (event) {
    if (event.type === 'deleted') {                   // 如果一个文件
      delete cached.caches.scripts[event.path];       // gulp-cache
      remember.forget('scripts', event.path);         // gulp-remer
    }
  });
});
```

将 buffer 变为 stream (内存中的内容)

有时候，你会需要这样一个 stream，它们的内容保存在一个变量中，而不是在一个实际的文件中。换言之，怎么不使用 `gulp.src()` 而创建一个 'gulp' stream。

我们来举一个例子，我们拥有一个包含 js 库文件的目录，以及一个包含一些模块的不同版本文件的目录。编译的目标是为每个版本创建一个 js 文件，其中包含所有库文件以及相应版本的模块文件拼接后的结果。

逻辑上我们将把这个拆分为如下步骤：

- 载入库文件
- 拼接库文件的内容
- 载入不同版本的文件
- 对于每个版本的文件，将其和库文件的内容拼接
- 对于每个版本的文件，将结果输出到一个文件

想象如下的文件结构：

```
├── libs
│   ├── lib1.js
│   └── lib2.js
└── versions
    ├── version.1.js
    └── version.2.js
```

你应该要得到这样的结果：

```
└── output
    ├── version.1.complete.js # lib1.js + lib2.js + version.1.js
    └── version.2.complete.js # lib1.js + lib2.js + version.2.js
```

一个简单的模块化处理方式将会像下面这样：

```
var gulp = require('gulp');
var runSequence = require('run-sequence');
var source = require('vinyl-source-stream');
var vinylBuffer = require('vinyl-buffer');
var tap = require('gulp-tap');
var concat = require('gulp-concat');
var size = require('gulp-size');
var path = require('path');
var es = require('event-stream');

var memory = {}; // 我们会将 assets 保存到内存中
```

```
// 载入内存中文件内容的任务
gulp.task('load-lib-files', function() {
  // 从磁盘中读取库文件
  return gulp.src('src/libs/*.js')
  // 将所有库文件拼接到一起
  .pipe(concat('libs.concat.js'))
  // 接入 stream 来获取每个文件的数据
  .pipe(tap(function(file) {
    // 保存文件的内容到内存
    memory[path.basename(file.path)] = file.contents.toString();
  }));
});

gulp.task('load-versions', function() {
  memory.versions = {};
  // 从磁盘中读取文件
  return gulp.src('src/versions/version.*.js')
  // 接入 stream 来获取每个文件的数据
  .pipe( tap(function(file) {
    // 在 assets 中保存文件的内容
    memory.versions[path.basename(file.path)] = file.contents.toStr
  }));
});

gulp.task('write-versions', function() {
  // 我们将不容版本的文件的名字保存到一个数组中
  var availableVersions = Object.keys(memory.versions);
  // 我们创建一个数组来保存所有的 stream 的 promise
  var streams = [];

  availableVersions.forEach(function(v) {
    // 以一个假文件名创建一个新的 stream
    var stream = source('final.' + v);
    // 从拼接后的文件中读取数据
    var fileContents = memory['libs.concat.js'] +
      // 增加版本文件的数据
      '\n' + memory.versions[v];

    streams.push(stream);

    // 将文件的内容写入 stream
    stream.write(fileContents);

    process.nextTick(function() {
      // 在下一次处理循环中结束 stream
      stream.end();
    });

    stream
    // 转换原始数据到 stream 中去，到一个 vinyl 对象/文件
    .pipe(vinylBuffer())
    //.pipe(tap(function(file) { /* 这里可以做一些对文件内容的处理操作 *
    .pipe(gulp.dest('output'));
```

```
  });

  return es.merge.apply(this, streams);
});

//============================================ 我们的主任务
gulp.task('default', function(taskDone) {
  runSequence(
    ['load-lib-files', 'load-versions'],  // 并行载入文件
    'write-versions',  // 一旦所有资源进入内存便可以做写入操作了
    taskDone           // 完成
  );
});

//============================================ 我们的监控任务
// 只在运行完 'default' 任务后运行，
// 这样所有的资源都已经在内存中了
gulp.task('watch', ['default'], function() {
  gulp.watch('./src/libs/*.js', function() {
    runSequence(
      'load-lib-files',  // 我们只需要载入更改过的文件
      'write-versions'
    );
  });

  gulp.watch('./src/versions/*.js', function() {
    runSequence(
      'load-versions',  // 我们只需要载入更改过的文件
      'write-versions'
    );
  });
});
```

在 gulp 中运行 Mocha 测试

运行所有的测试用例

```
// npm install gulp gulp-mocha

var gulp = require('gulp');
var mocha = require('gulp-mocha');

gulp.task('default', function() {
  return gulp.src(['test/test-*.js'], { read: false })
    .pipe(mocha({
      reporter: 'spec',
      globals: {
        should: require('should')
      }
    }));
});
```

在文件改动时候运行 mocha 测试用例

```
// npm install gulp gulp-mocha gulp-util

var gulp = require('gulp');
var mocha = require('gulp-mocha');
var gutil = require('gulp-util');

gulp.task('mocha', function() {
    return gulp.src(['test/*.js'], { read: false })
        .pipe(mocha({ reporter: 'list' }))
        .on('error', gutil.log);
});

gulp.task('watch-mocha', function() {
    gulp.watch(['lib/**', 'test/**'], ['mocha']);
});
```

仅仅传递更改过的文件

默认情况下，每次运行时候所有的文件都会传递并通过整个管道。通过使用 gulp-changed 可以只让更改过的文件传递过管道。这可以大大加快连续多次的运行。

```
// npm install --save-dev gulp gulp-changed gulp-jscs gulp-uglify

var gulp = require('gulp');
var changed = require('gulp-changed');
var jscs = require('gulp-jscs');
var uglify = require('gulp-uglify');

// 我们在这里定义一些常量以供使用
var SRC = 'src/*.js';
var DEST = 'dist';

gulp.task('default', function() {
    return gulp.src(SRC)
        // `changed` 任务需要提前知道目标目录位置
        // 才能找出哪些文件是被修改过的
        .pipe(changed(DEST))
        // 只有被更改过的文件才会通过这里
        .pipe(jscs())
        .pipe(uglify())
        .pipe(gulp.dest(DEST));
});
```

从命令行传递参数

```
// npm install --save-dev gulp gulp-if gulp-uglify minimist

var gulp = require('gulp');
var gulpif = require('gulp-if');
var uglify = require('gulp-uglify');

var minimist = require('minimist');

var knownOptions = {
  string: 'env',
  default: { env: process.env.NODE_ENV || 'production' }
};

var options = minimist(process.argv.slice(2), knownOptions);

gulp.task('scripts', function() {
  return gulp.src('**/*.js')
    .pipe(gulpif(options.env === 'production', uglify())) // 仅在生
    .pipe(gulp.dest('dist'));
});
```

然后，通过如下命令运行 gulp：

```
$ gulp scripts --env development
```

只重新编译被更改过的文件

通过使用 `gulp-watch` :

```
var gulp = require('gulp');
var sass = require('gulp-sass');
var watch = require('gulp-watch');

gulp.task('default', function() {
  return gulp.src('sass/*.scss')
    .pipe(watch('sass/*.scss'))
    .pipe(sass())
    .pipe(gulp.dest('dist'));
});
```

每个文件夹生成单独一个文件

如果你有一整套的文件目录，并且希望执行相应的一套任务，比如...

```
/scripts
/scripts/jquery/*.js
/scripts/angularjs/*.js
```

...然后希望完成如下的结果h...

```
/scripts
/scripts/jquery.min.js
/scripts/angularjs.min.js
```

...你将会需要像下面所示的东西...

```
var fs = require('fs');
var path = require('path');
var merge = require('merge-stream');
var gulp = require('gulp');
var concat = require('gulp-concat');
var rename = require('gulp-rename');
var uglify = require('gulp-uglify');

var scriptsPath = 'src/scripts';

function getFolders(dir) {
    return fs.readdirSync(dir)
      .filter(function(file) {
        return fs.statSync(path.join(dir, file)).isDirectory();
      });
}

gulp.task('scripts', function() {
   var folders = getFolders(scriptsPath);

   var tasks = folders.map(function(folder) {
      // 拼接成 foldername.js
      // 写入输出
      // 压缩
      // 重命名为 folder.min.js
      // 再一次写入输出
      return gulp.src(path.join(scriptsPath, folder, '/*.js'))
        .pipe(concat(folder + '.js'))
        .pipe(gulp.dest(scriptsPath))
        .pipe(uglify())
        .pipe(rename(folder + '.min.js'))
        .pipe(gulp.dest(scriptsPath));
   });

   return merge(tasks);
});
```

注：

- `folders.map` - 在每一个文件夹中分别执行一次函数，并且返回异步 stream
- `merge` - 汇总 stream，并且在所有的 stream 都完成后完成

串行方式运行任务，亦即，任务依赖

默认情况下，任务会以最大的并发数同时运行 -- 也就是说，它会不做任何等待地将所有的任务同时开起来。如果你希望创建一个有特定顺序的串行的任务链，你需要做两件事：

- 给它一个提示，用以告知任务在什么时候完成，
- 而后，再给一个提示，用以告知某任务需要依赖另一个任务的完成。

举个例子，我们假设你有两个任务，"one" 和 "two"，并且你明确的希望他们就以这样的顺序运行：

1. 在任务 "one" 中，你添加的一个提示，来告知何时它会完成。你可以传入一个回调函数，然后在完成后执行回调函数，也可以通过返回一个 promise 或者 stream 来让引擎等待它们分别地被解决掉。

2. 在任务 "two" 中，你添加一个提示，来告知引擎它需要依赖第一个任务的完成。

因此，这个例子将会是像这样：

```
var gulp = require('gulp');

// 传入一个回调函数，因此引擎可以知道何时它会被完成
gulp.task('one', function(cb) {
    // 做一些事 -- 异步的或者其他任何的事
    cb(err); // 如果 err 不是 null 和 undefined，流程会被结束掉，'two'
});

// 标注一个依赖，依赖的任务必须在这个任务开始之前被完成
gulp.task('two', ['one'], function() {
    // 现在任务 'one' 已经完成了
});

gulp.task('default', ['one', 'two']);
// 也可以这么写：gulp.task('default', ['two']);
```

另一个例子，通过返回一个 stream 来取代使用回调函数的方法：

```
var gulp = require('gulp');
var del = require('del'); // rm -rf

gulp.task('clean', function(cb) {
    del(['output'], cb);
});

gulp.task('templates', ['clean'], function() {
    var stream = gulp.src(['src/templates/*.hbs'])
        // 执行拼接，压缩，等。
        .pipe(gulp.dest('output/templates/'));
    return stream; // 返回一个 stream 来表示它已经被完成

});

gulp.task('styles', ['clean'], function() {
    var stream = gulp.src(['src/styles/app.less'])
        // 执行一些代码检查，压缩，等
        .pipe(gulp.dest('output/css/app.css'));
    return stream;
});

gulp.task('build', ['templates', 'styles']);

// templates 和 styles 将会并行处理
// clean 将会保证在任一个任务开始之前完成
// clean 并不会被执行两次，尽管它被作为依赖调用了两次

gulp.task('default', ['build']);
```

拥有实时重载（live-reloading）和 CSS 注入的服务器

使用 BrowserSync 和 gulp，你可以轻松地创建一个开发服务器，然后同一个 WiFi 中的任何设备都可以方便地访问到。BrowserSync 同时集成了 live-reload 所以不需要另外做配置了。

首先安装模块：

```
$ npm install --save-dev browser-sync
```

然后，考虑拥有如下的目录结构...

```
gulpfile.js
app/
  styles/
    main.css
  scripts/
    main.js
  index.html
```

... 通过如下的 `gulpfile.js` ，你可以轻松地将 `app` 目录中的文件加到服务器中，并且所有的浏览器都会在文件发生改变之后自动刷新：

```
var gulp = require('gulp');
var browserSync = require('browser-sync');
var reload = browserSync.reload;

// 监视文件改动并重新载入
gulp.task('serve', function() {
  browserSync({
    server: {
      baseDir: 'app'
    }
  });

  gulp.watch(['*.html', 'styles/**/*.css', 'scripts/**/*.js'], {cwd
});
```

在 `index.html` 中引入 CSS：

```
<html>
  <head>
    ...
    <link rel="stylesheet" href="styles/main.css">
    ...
```

通过如下命令启动服务，并且打开一个浏览器，访问默认的 URL (http://localhost:3000)：

```
gulp serve
```

+ CSS 预处理器

一个常见的使用案例是当 CSS 文件文件预处理之后重载它们。以 sass 为例，这便是你如何指示浏览器无需刷新整个页面而只是重载 CSS。

考虑有如下的文件目录结构...

```
gulpfile.js
app/
  scss/
    main.scss
  scripts/
    main.js
  index.html
```

... 通过如下的 `gulpfile.js`，你可以轻松地监视 `scss` 目录中的文件，并且所有的浏览器都会在文件发生改变之后自动刷新：

```
var gulp = require('gulp');
var sass = require('gulp-ruby-sass');
var browserSync = require('browser-sync');
var reload = browserSync.reload;

gulp.task('sass', function() {
  return sass('scss/styles.scss')
    .pipe(gulp.dest('app/css'))
    .pipe(reload({ stream:true }));
});

// 监视 Sass 文件的改动，如果发生变更，运行 'sass' 任务，并且重载文件
gulp.task('serve', ['sass'], function() {
  browserSync({
    server: {
      baseDir: 'app'
    }
  });

  gulp.watch('app/scss/*.scss', ['sass']);
});
```

在 `index.html` 文件中引入预处理后的 CSS 文件：

```
<html>
  <head>
    ...
    <link rel="stylesheet" href="css/main.css">
    ...
```

通过如下命令启动服务，并且打开一个浏览器，访问默认的 URL (http://localhost:3000)：

```
gulp serve
```

附注：

- 实时重载（Live reload），CSS 注入以及同步滚动可以在 BrowserStack 虚拟机里无缝执行。
- 设置 `tunnel: true` 来使用一个公开的 URL 来访问你本地的站点 (支持所有 BrowserSync 功能)。

通过 stream 工厂来共享 stream

如果你在多个任务中使用了相同的插件，你可能发现你很想把这些东西以 DRY 的原则去处理。这个方法可以创建一些工厂来把你经常使用的 stream 链分离出来。

我们将使用 lazypipe 来完成这件事。

这是我们的例子：

```
var gulp = require('gulp');
var uglify = require('gulp-uglify');
var coffee = require('gulp-coffee');
var jshint = require('gulp-jshint');
var stylish = require('jshint-stylish');

gulp.task('bootstrap', function() {
  return gulp.src('bootstrap/js/*.js')
    .pipe(jshint())
    .pipe(jshint.reporter(stylish))
    .pipe(uglify())
    .pipe(gulp.dest('public/bootstrap'));
});

gulp.task('coffee', function() {
  return gulp.src('lib/js/*.coffee')
    .pipe(coffee())
    .pipe(jshint())
    .pipe(jshint.reporter(stylish))
    .pipe(uglify())
    .pipe(gulp.dest('public/js'));
});
```

然后，使用了 lazypipe 之后，将会是这样：

```
var gulp = require('gulp');
var uglify = require('gulp-uglify');
var coffee = require('gulp-coffee');
var jshint = require('gulp-jshint');
var stylish = require('jshint-stylish');
var lazypipe = require('lazypipe');

// 赋给 lazypipe
var jsTransform = lazypipe()
  .pipe(jshint)
  .pipe(jshint.reporter, stylish)
  .pipe(uglify);

gulp.task('bootstrap', function() {
  return gulp.src('bootstrap/js/*.js')
    .pipe(jsTransform())
    .pipe(gulp.dest('public/bootstrap'));
});

gulp.task('coffee', function() {
  return gulp.src('lib/js/*.coffee')
    .pipe(coffee())
    .pipe(jsTransform())
    .pipe(gulp.dest('public/js'));
});
```

你可以看到，我们把多个任务中都在使用的 JavaScript 管道（JSHint + Uglify）分离到了一个工厂。工厂可以在任意多的任务中重用。你也可以嵌套这些工厂，或者把它们连接起来，已达到更好的效果。分离出每个共享的管道，也可以让你能够集中地管理，当你的工作流程更改后，你只需要修改一个地方即可。

指定一个新的 cwd (当前工作目录)

在一个多层嵌套的项目中，这是非常有用的，比如：

```
/project
  /layer1
  /layer2
```

你可以使用 gulp 的 CLI 参数 `--cwd`.

在 `project/` 目中中：

```
gulp --cwd layer1
```

如果你需要对特定的匹配指定一个 cwd，你可以使用 glob-stream 的 `cwd` 选项：

```
gulp.src('./some/dir/**/*.js', { cwd: 'public' });
```

分离任务到多个文件中

如果你的 `gulpfile.js` 开始变得很大，你可以通过使用 require-dir 模块将任务分离到多个文件

想象如下的文件结构：

```
gulpfile.js
tasks/
├── dev.js
├── release.js
└── test.js
```

安装 `require-dir` 模块：

```
npm install --save-dev require-dir
```

在 `gulpfile.js` 中增加如下几行代码：

```
var requireDir = require('require-dir');
var dir = requireDir('./tasks');
```

使用外部配置文件

这有很多好处，因为它能让任务更加符合 DRY 原则，并且 config.json 可以被其他的任务运行器使用，比如 `grunt` 。

`config.json`

```
{
  "desktop" : {
    "src" : [
      "dev/desktop/js/**/*.js",
      "!dev/desktop/js/vendor/**"
    ],
    "dest" : "build/desktop/js" },
  "mobile" : {
    "src" : [
      "dev/mobile/js/**/*.js",
      "!dev/mobile/js/vendor/**"
    ],
    "dest" : "build/mobile/js" } }
```

`gulpfile.js`

```
// npm install --save-dev gulp gulp-uglify
var gulp = require('gulp');
var uglify = require('gulp-uglify');
var config = require('./config.json');

function doStuff(cfg) {
  return gulp.src(cfg.src)
    .pipe(uglify())
    .pipe(gulp.dest(cfg.dest));
}

gulp.task('dry', function() {
  doStuff(config.desktop);
  doStuff(config.mobile);
});
```

在一个任务中使用多个文件来源

```
// npm install --save-dev gulp merge-stream

var gulp = require('gulp');
var merge = require('merge-stream');

gulp.task('test', function() {
  var bootstrap = gulp.src('bootstrap/js/*.js')
    .pipe(gulp.dest('public/bootstrap'));

  var jquery = gulp.src('jquery.cookie/jquery.cookie.js')
    .pipe(gulp.dest('public/jquery'));

  return merge(bootstrap, jquery);
});
```

`gulp.src` 会以文件被添加的顺序来 emit：

```
// npm install gulp gulp-concat

var gulp = require('gulp');
var concat = require('gulp-concat');

gulp.task('default', function() {
  return gulp.src(['foo/*', 'bar/*'])
    .pipe(concat('result.txt'))
    .pipe(gulp.dest('build'));
});
```

Browserify + Uglify2 和 sourcemaps

Browserify 现在已经成为了一个不可或缺的重要工具了，然后要让它能完美的和 gulp 一起协作，还得需要做一些封装处理。先面便是一个使用 Browserify 并且增加一个完整的 sourcemap 来对应到单独的每个源文件。

同时请看: 组合 Streams 来处理错误 范例来查看如何处理你的 stream 中 browserify 或者 uglify 的错误。

```
'use strict';

var browserify = require('browserify');
var gulp = require('gulp');
var source = require('vinyl-source-stream');
var buffer = require('vinyl-buffer');
var uglify = require('gulp-uglify');
var sourcemaps = require('gulp-sourcemaps');
var gutil = require('gulp-util');

gulp.task('javascript', function () {
  // 在一个基础的 task 中创建一个 browserify 实例
  var b = browserify({
    entries: './entry.js',
    debug: true
  });

  return b.bundle()
    .pipe(source('app.js'))
    .pipe(buffer())
    .pipe(sourcemaps.init({loadMaps: true}))
        // 在这里将转换任务加入管道
        .pipe(uglify())
        .on('error', gutil.log)
    .pipe(sourcemaps.write('./'))
    .pipe(gulp.dest('./dist/js/'));
});
```

Browserify + Globs

Browserify + Uglify2 展示了如何设置一个基础的 gulp 任务来把一个 JavaScript 文件以及它的依赖打包，并且使用 UglifyJS 压缩并且保留 source map。 然而这还不够，这里还将会展示如何使用 gulp 和 Browserify 将多个文件打包到一起。

同时请看: 组合 Streams 来处理错误 范例来查看如何处理你的 stream 中 browserify 或者 uglify 的错误。

```
'use strict';

var browserify = require('browserify');
var gulp = require('gulp');
var source = require('vinyl-source-stream');
var buffer = require('vinyl-buffer');
var globby = require('globby');
var through = require('through2');
var gutil = require('gulp-util');
var uglify = require('gulp-uglify');
var sourcemaps = require('gulp-sourcemaps');
var reactify = require('reactify');

gulp.task('javascript', function () {
  // gulp 希望任务能返回一个 stream，因此我们在这里创建一个
  var bundledStream = through();

  bundledStream
    // 将输出的 stream 转化成为一个包含 gulp 插件所期许的一些属性的 stream
    .pipe(source('app.js'))
    // 剩下的部分，和你往常缩写的一样。
    // 这里我们直接拷贝 Browserify + Uglify2 范例的代码。
    .pipe(buffer())
    .pipe(sourcemaps.init({loadMaps: true}))
      // 在这里将相应 gulp 插件加入管道
      .pipe(uglify())
      .on('error', gutil.log)
    .pipe(sourcemaps.write('./'))
    .pipe(gulp.dest('./dist/js/'));

  // "globby" 替换了往常的 "gulp.src" 为 Browserify
  // 创建的可读 stream。
  globby(['./entries/*.js'], function(err, entries) {
    // 确保任何从 globby 发生的错误都被捕获到
    if (err) {
      bundledStream.emit('error', err);
      return;
    }

    // 创建 Browserify 实例
```

```
    var b = browserify({
      entries: entries,
      debug: true,
      transform: [reactify]
    });

    // 将 Browserify stream 接入到我们之前创建的 stream 中去
    // 这里是 gulp 式管道正式开始的地方
    b.bundle().pipe(bundledStream);
  });

  // 最后，我们返回这个 stream，这样 gulp 会知道什么时候这个任务会完成
  return bundledStream;
});
```

同时输出一个压缩过和一个未压缩版本的文件

同时输出压缩过的和未压缩版本的文件可以通过使用 `gulp-rename` 然后 pipe 到 `dest` 两次来实现 (一次是压缩之前的，一次是压缩后的)：

```
'use strict';

var gulp = require('gulp');
var rename = require('gulp-rename');
var uglify = require('gulp-uglify');

var DEST = 'build/';

gulp.task('default', function() {
  return gulp.src('foo.js')
    // 这会输出一个未压缩过的版本
    .pipe(gulp.dest(DEST))
    // 这会输出一个压缩过的并且重命名未 foo.min.js 的文件
    .pipe(uglify())
    .pipe(rename({ extname: '.min.js' }))
    .pipe(gulp.dest(DEST));
});
```

改变版本号以及创建一个 git tag

如果你的项目遵循语义化版本，那么，把那些发布新版本的时候需要做的事情通过自动化的手段去完成将会是个很不错的主意。 下面有一个简单的范例展示了如何改变项目的版本号，将更新提交到 git，以及创建一个 tag。

```
 var gulp = require('gulp');
var runSequence = require('run-sequence');
var bump = require('gulp-bump');
var gutil = require('gulp-util');
var git = require('gulp-git');
var fs = require('fs');

gulp.task('bump-version', function () {
// 注意：这里我硬编码了更新类型为 'patch'，但是更好的做法是用
//       minimist (https://www.npmjs.com/package/minimist) 通过检测一
//       一个 'major'， 'minor' 还是一个 'patch'。
  return gulp.src(['./bower.json', './package.json'])
    .pipe(bump({type: "patch"}).on('error', gutil.log))
    .pipe(gulp.dest('./'));
});

gulp.task('commit-changes', function () {
  return gulp.src('.')
    .pipe(git.commit('[Prerelease] Bumped version number', {args:
});

gulp.task('push-changes', function (cb) {
  git.push('origin', 'master', cb);
});

gulp.task('create-new-tag', function (cb) {
  var version = getPackageJsonVersion();
  git.tag(version, 'Created Tag for version: ' + version, function
    if (error) {
      return cb(error);
    }
    git.push('origin', 'master', {args: '--tags'}, cb);
  });

  function getPackageJsonVersion () {
    // 这里我们直接解析 json 文件而不是使用 require，这是因为 require 会缓
    return JSON.parse(fs.readFileSync('./package.json', 'utf8')).ve
  };
});

gulp.task('release', function (callback) {
  runSequence(
    'bump-version',
```

```
    'commit-changes',
    'push-changes',
    'create-new-tag',
    function (error) {
      if (error) {
        console.log(error.message);
      } else {
        console.log('RELEASE FINISHED SUCCESSFULLY');
      }
      callback(error);
    });
});
```

Swig 以及 YAML front-matter 模板

模板可以使用 `gulp-swig` 和 `gulp-front-matter` 来设置：

`page.html`

```
---
title: Things to do
todos:
    - First todo
    - Another todo item
    - A third todo item
---
<html>
    <head>
        <title>{{ title }}</title>
    </head>
    <body>
        <h1>{{ title }}</h1>
        <ul>{% for todo in todos %}
          <li>{{ todo }}</li>
        {% endfor %}</ul>
    </body>
</html>
```

`gulpfile.js`

```
var gulp = require('gulp');
var swig = require('gulp-swig');
var frontMatter = require('gulp-front-matter');

gulp.task('compile-page', function() {
  gulp.src('page.html')
      .pipe(frontMatter({ property: 'data' }))
      .pipe(swig())
      .pipe(gulp.dest('build'));
});

gulp.task('default', ['compile-page']);
```